Fall 10-1-1989

APPL/A: A Prototype Language for Software Process Programming ; CU-CS-448-89

Stanley M. Sutton Jr.
University of Colorado Boulder

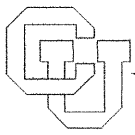
Dennis M. Heimbigner
University of Colorado Boulder

Leon J. Osterweil
University of Colorado Boulder

APPL/A: A Prototype Language for Software Process Programming

Stanley M. Sutton, Jr.
Dennis Heimbigner
Leon J. Osterweil

CU-CS-448-89



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

APPL/A: A Prototype Language for Software Process Programming

Stanley M. Sutton, Jr. Dennis Heimbigner
Leon J. Osterweil*

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
October, 1989
CU-CS-448-89

Abstract

Software process programming is the codification of software development processes in a formal programming language. The premise of software process programming is that software processes are themselves software. Its goal is to enable software processes to be programmed using the methods, tools, and techniques that have been developed for conventional software.

APPL/A is a prototype process programming language that explores issues in software object management. Important issues in software object management include persistent data, concurrency control, complex relationships among objects, associative access, and consistency management, among others.

APPL/A is an extension to Ada. Ada provides general programming-language capabilities. APPL/A extensions include three new program units: relations, predicates, and triggers. Relations provide abstract persistent storage with programmable implementations. Relation attributes may be composite and derived. Triggers react to operations on relations. Predicates specify the consistency of relations and can be optionally enforced like constraints. Other APPL/A extensions include five compound transaction-like statements that support consistency management.

The APPL/A extensions to Ada provide capabilities in several areas that are essential to software object management. These include persistence data and programmable implementations for persistent storage, representation of relationships among objects, derived data, queries, activity and inferencing, transactions, and a flexible model of consistency. APPL/A has been used in several prototype process programs, and earlier versions of the language have been refined based on this experience.

*Department of Information and Computer Science, University of California, Irvine, CA 92717

1 Introduction

Software process programming is the codification of software development processes in a formal programming language [20]. A process program can in principle represent and implement the structure of software products and the processes by which they are constructed. Software process programming is based on the premise that software processes are themselves software.

The advantages of the formalization of software processes and products are potentially the same as those for the formalization of software in general: software process programs should enable software processes to be automatically executed, tested, analyzed, debugged, revised, maintained, reused, and communicated more effectively. Additionally, development methodologies that have been devised for conventional software should be applicable to software process programs. For example, process programs can be coded in accordance with a design that is developed to address specified requirements for the processes.

Realizing these potential advantages requires suitable software process programming languages (PPLs). PPLs should include constructs and capabilities that are appropriate for the domain of software processes, but the detailed requirements for such languages are difficult to determine *a priori*. Software processes in general are not as well understood as conventional application domains, and experience in programming software processes is comparatively very limited. It seems reasonable to assume that PPLs must subsume the capabilities of conventional programming languages. Thus we would expect PPLs to include such things as variables, subroutines, and abstract data types. However, it is expected that PPLs must also include extensions and specializations that reflect the distinctive aspects of software processes and products.

Unfortunately, the present state of process programming research involves a circular dependency. Knowledge of software processes will increase when they can be programmed effectively, but the effective programming of software processes depends on the availability of appropriate PPLs, and the requirements for PPLs cannot be confidently specified until we have more knowledge of software processes. Breaking this cycle requires the design and implementation of *prototype* PPLs and the use of these languages to develop

prototype process programs. In turn, knowledge gained from these prototypes can be used to refine the next generation of languages and programs, and the prototyping exercise can be reiterated.

In this paper we present APPL/A [34], a prototype PPL based on Ada [1]. APPL/A is designed to explore a particular aspect of process programming, namely the problems of object management in software processes. Object management is an important area of requirements for PPLs [16,23,22,35,9]. Recognized problems in software object management include

- The need to manage many kinds of persistent objects, including environment components (e.g. tools) and development products (e.g. requirements, design, and code).
- The need to support concurrent processes which require shared access to objects.
- The need to maintain a wide variety of relationships among objects, including derivation and other dependencies.
- The need for associative access to support *ad hoc* queries about process and product state.
- The need accommodate changing standards of consistency during long and complicated processes.

The APPL/A design is intended to explore these and related issues.

APPL/A research is one part of the process programming research that is taking place in the Arcadia project [36]. Other research in Arcadia encompasses additional aspects of software processes (such as data modeling and process organization) and includes the development of prototype process programs.

This paper is organized as follows. Section 2 presents our approach to prototyping a process programming language. Section 3 describes our design of the prototype PPL APPL/A. Section 4 provides an overview of APPL/A [34], including some examples of APPL/A constructs. The paper concludes with a summary of our experience and the status of work (Section 5).

2 Prototyping Approach

Software object management is the management of the software artifacts comprising software environments and products. These include such things as tools, libraries of reusable components, product requirements and designs, code of various kinds, test programs, data, and results, and project management data. Software object management is itself a complicated multi-faceted problem involving issues ranging from high-level data abstractions to low-level implementation mechanisms. For example, software objects must be stored persistently, i.e. between program executions, and access to shared objects by concurrent processes must be controlled. Some objects will be created manually, while others will be derived automatically. Derivation dependencies among objects must be kept up-to-date as needed, and objects in general must be kept consistent with constraints imposed by process and product requirements. Consistency must be managed in an environment in which processes are prone to failure and restart and in which concurrent processes may have conflicting assumptions and goals.

Our premise is that a multi-faceted problem like software object management requires a multi-faceted solution. It must be addressed in terms of a broad range of features and capabilities. We identified the following areas for support in our prototype PPL:

- **Persistence and abstraction**: Software processes involve the creation, use, and management of persistent objects, i.e. objects that outlive the processes in which they are used. These objects include environment components and intermediate and final development products. In recent years several "persistent programming languages" have been developed to facilitate the development of applications that involve persistent data. These languages provide integrated support for persistent data: persistent data are represented in the same type system as transient data, and operational boundaries between persistent and transient data are reduced. Applications are simplified to the extent that persistent and transient data can be treated similarly. Examples of persistent programming languages include PS-Algol [4,14], Adaplex [29], Owl [28,27] (the language of the Trellis environment), E [25] (the database

implementation language of EXODUS [11]), and COP [3] (the data-manipulation language of the VBase environment). The motivation for persistent programming languages in general is presented in more detail in [4]. Issues in the design of persistent programming languages are discussed in [19,33,6,5]. Because software processes necessarily deal with persistent objects we believe that software process programming languages should be persistent programming languages.

A limitation of many persistent programming languages (e.g. Adaplex and PS-Algol) is that they rely on a fixed persistent object management system (OMS) for storage of persistent objects. No single OMS is likely to best meet all of the needs of large-scale software projects. Such projects often involve persistent data stored in a variety of systems, a given OMS may evolve over time, and the OMS itself may be a research or development product. A related issue is that a fixed OMS may imply fixed strategies for the computation and caching of derived objects. An example is the Odin OMS [12,13], which uses a "lazy" evaluation strategy with caching of derived objects. These strategies work well in many situations, but other strategies (e.g. "eager" or opportunistic derivations, no caching of derived objects) may be most appropriate in others. In light of the limitations imposed by a fixed OMS, we believe that the implementation of persistent storage and related capabilities should be *programmable.* The programmer should be able to specify how persistent storage is implemented and how derived objects are computed and cached.

- **Relations and derived data**: An important aspect of software object management is the variety of relationships among objects that must be represented and maintained [35]. Prominent among these are derivation relationships in which derived objects are computed by some software tool or process from given objects (which may themselves be derived). A typical example is the derivation relationship between object code and the source code from which it is compiled. Many other kinds of relationships also occur among software objects. For example, source code is related to configurations, designs are related to requirements, test cases are related to acceptance criteria, programmers are

related to projects, and so on. Some generic structures are needed to represent the relationships among individual objects and to group these representations where they are associated. Moreover, the abstract representation of relationships should be tied to implementation mechanisms that ensure that the relationships are maintained as needed in a consistent and up-to-date state.

- **Queries**: Software process management and control depend on the state of the environment and the products in it. The efficient execution of these tasks depends on efficient access to information about the state. It is inadequate to obtain needed information by exhaustive traversal of data structures. Instead, some means are needed to allow information to be obtained associatively, i.e. by values associated with the objects or data sought. For example, it should be possible to retrieve the source code modules associated with project "X" by asking for the source code modules associated with project "X" rather than by iterating through all source code modules in the environment. In other words, what is needed is support for queries on data.

- **Activity and triggering**: One of the goals of software process programming is increased automation of development processes. To the extent that processes can be automated the problems associated with the manual execution of processes are reduced. The benefits should be increased consistency, correctness, and efficiency. The kinds of processes that may be automatable include the creation of objects, the propagation of changes to update outdated dependency relationships, and the repair of consistency violations (among others) [35]. The semantics of these processes are all application dependent, but a PPL should provide mechanisms that enable them to be conveniently programmed. These mechanisms should include concurrency and both *pro-active* and *re-active* processes. The latter correspond to daemons or triggers which respond to state changes or operations in the environment (for example, AP5 [15] includes trigger rules that initiate actions in response to state changes, while VBase [3] supports triggers that react to operations on objects). These mechanisms should be composable in various ways to allow the chaining of processes necessary to support various kinds of inferencing.

- **Transactions**: Software processes can involve substantial concurrency. If concurrent processes share objects then some transaction-like mechanism must be provided for concurrency control to prevent conflicting access. Some of issues involved in concurrency control for software processes are discussed in [22].

- **Flexible model of consistency**: Consistency in software processes is relative and evolving. Constraints that must characterize a final software product may not all be known in advance, and known final constraints may not be relevant during all phases of development. Moreover, different software processes may have different preconditions or goals, and these may conflict. Consequently, inconsistency in software products is virtually inevitable. Inconsistency may arise as new constraints are introduced or as one process updates objects in a way that conflicts with the preconditions or goals of another. Inconsistency may also arise because of failures in development processes. Failure is relatively common in software processes compared to transactions in conventional databases. Software processes are long and complicated, subject to frequent interruptions, and often nondeterministic and tentative. Their effects on objects are thus relatively likely to be inconsistent, incorrect, or incomplete. A software process programming language should provide a flexible model of consistency for software objects that accommodates these conditions.

This list could be extended, for example, it addresses data modeling in a limited way and it does not address security at all. Nevertheless, we felt that it would be a sufficient challenge to design of a language with support in just the areas listed. Additionally, we believe that integrated support in these areas can significantly enhance software object management. Thus features and capabilities comprise the requirements for the design of our prototype PPL.

3 Language Design

Our overall goal in developing APPL/A is to explore issues in the design and integration of language features to support software object management. APPL/A is defined as an extension to Ada [1]. We chose an existing language as the basis of our prototype for two reasons. First it would allow us to focus our attention on features of real concern to us. Second, it would enable us to take advantage of existing language technology.

Our choice of Ada was motivated by both technical and pragmatic considerations. Ada provides constructs, for example packages and tasks, which have served as models for our extensions. Packages support abstraction and information hiding, which is advantageous in the design of persistent objects with programmable implementations. Tasks support concurrency, which is essential to our requirements for activity and triggering. We have also found these and other aspects of Ada (e.g. the composition of programs from separately compilable program units) useful in the formulation of prototype process programs. Additionally, our work is intended to draw on and support Ada-related work in the Arcadia project [36].

It should be admitted that the choice of Ada is open to question in that Ada lacks certain features and capabilities that might be useful. We have in fact found Ada limiting in several respects, including the lack of subprogram and package variables, lack of finalization code for packages, and inflexibility in the exception-handling mechanism. Nevertheless, Ada has provided an effective basis on which to build our prototype and explore the object-management and language design issues in which we are interested.

The focus of our extensions to Ada are *relation* units, which are effectively abstract, persistent, and active multisets of *tuples*. Tuples are types like records. Relations are comparable to a combination of packages and tasks. They define and group information like packages, and they represent concurrent threads of control like tasks. Like both packages and tasks relations have separate specifications and bodies. The specifications stipulate the semantics of the relations. The semantics include the tuple type stored in the relation, automatically derived (i.e. computed) attributes, dependency specifications (which indicate how derived attributes are to be computed). Relation specifications also indicate the operations available on the

relation, which may include the insertion, deletion, update, and retrieval of tuples; these are represented as entries. The deletion, update, and retrieval of tuples are all selective (i.e. associative) operations, thus they support an important form of query. Relation bodies provide a context in which the implementation of those semantics can be programmed. Relation bodies must provide persistent storage for tuples, compute derived attributes as specified and keep them up-to-date, and implement the relation operations. Relations thus integrate several required capabilities: representation of relationships among objects, derived data, queries, persistence, programmable implementations, and activity.

The other principal extensions to Ada are centered around relations. APPL/A includes two other new program units: predicates and triggers. Predicates are boolean expressions over relations; predicates can be tested like functions and (optionally) enforced like constraints. Triggers are concurrent but reactive threads of control that respond automatically to signals generated by operations on relations. APPL/A also includes several new control constructs. Chief among these are several block-like statements that provide capabilities such as concurrency control and atomicity which are ordinarily associated with transactions in conventional databases. These additional units and constructs address requirements in the areas of activity and triggering and a flexible approach to consistency management.

We have chosen to focus on relations because they provide an effective integrating mechanism for the broad range of features and capabilities that we feel are essential to software object management. Our prototyping effort is simplified to the extent that we don't have to provide these capabilities for all types in a language (e.g. predicates over arbitrary types). At the same time, having brought so many features and capabilities to bear on relations, we are still able to investigate their combinations and interactions, and this is a principal goal of the prototyping exercise. Moreover, relations (as we have defined them) are a powerful and generic construct which should be widely useful in process programs, especially in conjunction with the related capabilities such as predicates and triggers. Thus we believe that the focus on relations should not be a severe restriction for application programming.

A detailed discussion of the APPL/A relational model, our rationale for the use of relations, and a comparison with other data models is found in [35]. Some other recent projects in which relations are used include the advanced data-management system Postgres [26,30] and AP5, which extends Common Lisp with relations [15].

4 Overview of APPL/A

This section presents some of the details of the APPL/A extensions to Ada and shows some examples of APPL/A constructs. A thorough presentation of APPL/A is beyond the scope of this paper. Relations and triggers are emphasized here. A brief introduction to predicates and related consistency-management constructs is also presented.

4.1 Relations

Relations are a special kind of program unit that provides for the persistent storage of data in the form of tuples. An APPL/A relation is syntactically similar to a combination of Ada packages and tasks. Each relation has a specification and a body. A typical specification, for relation `Word_Count`, is shown in Figure 1. This relation represents the derivation relationship between the input and output of a "word-count" tool which computes the numbers of lines, words, and characters in a given text object. The features of APPL/A relations are explained below in terms of this example.

Each relation has a defining tuple type that determines the names and types of attributes for the relation. A tuple type is similar to a record type, but the attributes have modes like Ada parameters. The attribute modes indicate the way in which attributes may take on values:

- Attributes of mode **in** must be given values directly by the user.
- Attributes of mode **out** must take on computed values and cannot take on values given by the user.

```
with WC;    - separately defined word-count tool
with text_def; use text_def;


Relation Word_Count is
    type wc_tuple is tuple
        text: in text_type;
        lines, words, characters: out natural;
    end tuple;
entries
    entry insert(text: in text_type);
    entry delete(t: in wc_tuple);
    entry find(iterator: in out integer => 0;
        first: in boolean => true;
        found: out boolean => false;
        t: out wc_tuple;
        select_text: boolean => false; text: text_type;
        select_lines: boolean => false; lines: natural;
        select_words: boolean => false; words: natural;
        select_characters: boolean => false; characters: natural);
dependencies
    determine lines, words, characters
        by wc(text, lines, words, characters);
End Word_Count;
```

Figure 1: Specification for Relation Word_Count

- Attributes of mode **in out** may take on given and computed values in turn.

Thus, attributes of modes **in out** and **out** represent derived data. Attributes may also have composite values (i.e. they are not restricted to atomic values like conventional relations). In `Word_Count` the value of the attribute `text` must be given, while the values of the attributes `lines`, `words`, and `characters` are computed automatically.

The operations on a relation are represented by entries analogous to task entries. The entries for a relation must be some non-empty subset of `insert`, `update`, `delete`, and `find`. For example, `Word_Count` includes entries `insert`, `delete`, and `find`, but not `update`.

The `insert` entry takes parameters for attributes of mode **in** and **in out** and implements the insertion of a tuple with those parameters into the relation. The `update` entry enables a given tuple in the relation to

be assigned new values for attributes of mode **in** and **in out**. The **delete** entry deletes a given tuple from the relation. The **find** entry returns one tuple selected by a list of given attribute values. The **find** entry is also used to automatically construct iterators for the relations and to automatically implement predefined functions **tuple** and **member** (which respectively retrieve selected tuples from a relation and test given tuples for membership in a relation).

The specification of any relation with derived attributes may also contain a *dependency specification*, which indicates how the derived attributes are to be computed. In **Word_Count** the dependency specification stipulates that for each tuple the attributes **lines**, **words**, and **characters** are to be computed by a call to the predefined procedure **WC** given the corresponding value of attribute **text** as input. In this way **Word_Count** represents the derivation relationship established by the **WC** tool between text objects and the counts of lines, words, and characters computed by this tool. It is the responsibility of the body of the relation to automatically carry out the computations necessary to assign values to derived attributes. If a relation has a dependency specification then the computation of attributes must be carried out according to that specification. Computed attribute values are required to be kept up-to-date with respect to the values of other attributes from which they are derived. In particular, the values of derived attributes in each tuple returned by a call to **find** must be computed subsequent to the most recent **insert** or **update** operation on that tuple.

Another example, the specification for relation **Source_to_Object**, is shown in Figure 2. This relation represents the derivation relationship between source code and the object code compiled from it.

It is the responsibility of the body of a relation to implement the semantics of that relation. This means that the relation body must

- provide persistent storage;
- implement the relation entries;

```
with compile;      – separately defined compiler
with Code_Types; use Code_Types;


Relation Source_to_Object is
    type src_to_obj_tuple is tuple
        src: in source_code;
        obj: out object_code;
    end tuple;
entries
    entry insert ( src: in source_code);
    entry update (t: in src_to_obj_tuple;
                  update_src: in boolean => false;
                  src: in source_code);
    entry delete (t: in src_to_obj_tuple);
    entry find (iterator: in out integer => 0;
                  first: in boolean => true;
                  found: out boolean => false;
                  t: out src_to_obj_tuple;
                  select_src: in boolean => false;
                  src: in source_code);
dependencies
    determine obj by compile(src, obj);
end Source_to_Object;
```

Figure 2: Specification for Relation Source_to_Object

- compute and assign values for derived attributes.

However, the implementation of a relation can be left up to the programmer of the relation (although a default implementation mechanism will be available). In this respect APPL/A relations are *programmable.* Apart from the requirements listed above, the implementation of a relation is *not* constrained with respect to

- the persistent storage system;
- the derivation strategy for computed attributes (e.g. eager, opportunistic, or lazy);
- the caching strategy for computed attributes (e.g. cached when computed or recomputed when needed);

The implementor of a relation can program the body in any appropriate way that satisfies the required

semantics, and the implementation can change over time without affecting users of the relation (who rely on the specification). In this way APPL/A allows relation implementations to be customized for particular projects and installations. The ability to program implementations also enables foreign tools and data to be imported and encapsulated behind a standardized relational interface.

4.2 Triggers

Triggers are like tasks in that they represent concurrent threads of control. However, triggers differ from tasks in that triggers lack entries. Instead, triggers *react* automatically to operations on relations.

Triggers have only one part, i.e. they do not have separate specifications and bodies. Triggers lack a separate specification part because they do not declare anything that can be referenced by other program units. (Because triggers are strictly reactive they do not have an operational interface, and their intended role does not require them to make declarations of any other kind.) Syntactically a trigger comprises a loop over a *selective trigger* statement. A selective trigger statement is analogous to an Ada selective wait statement, except that it has **upon** alternatives instead of accept alternatives. Each upon alternative consists of an **upon** statement followed by a (possibly empty) sequence of statements. The **upon** statements identify the relation operations to which a response is to be made. The statements within and following the **upon** statement encode the trigger's response to the relation operation.

The trigger `Maintain_Source_WC` is shown in Figure 3. The goal of the trigger is to automatically collect and maintain up-to-date word-count data for source-code modules. This trigger responds to operations on relation `Source_to_Object` (Figure 2). The trigger propagates changes in `Source_to_Object` to the relation `Word_Count`. For example, when a new source-code object is inserted into `Source_to_Object`, the trigger automatically inserts that object into `Word_Count`; the trigger makes analogous responses to update and delete operations on `Source_to_Object`. (For simplicity, this example assumes that source modules of interest are stored in `Source_to_Object`, so the trigger responds to operations on that relation. In a more

realistic system source modules would probably be stored in a separate `Source_Code` relation, along with additional information such as author, permissions, timestamps, etc. In that case a trigger could be defined to respond to operations on the `Source_Code` relation.)

Trigger `Maintain_Source_WC` includes three **upon** statements, one each for the insert, update, and delete entries of `Source_to_Object`. Each of these **upon** statements is for a *completion* event, i.e. a response is to be triggered only upon the successful completion of the corresponding entry call. (**Upon** statements can also designate *acceptance* events, in which case a response would be triggered by the acceptance of the relation entry call.) Each **upon** statement also includes a list of formal parameters. For an acceptance event these comprise the **in** parameters for the relation entry call; for a completion event these comprise both the **in** and **out** parameters for the call. Through these parameters the actual values given to and returned from the relation entry call are made available to the trigger. Although it is not shown in the example, **upon** statements may also be given priority values. When an event occurs (i.e. a relation entry call is accepted or completed), a signal is sent to each trigger that designates that event in an **upon** statement. This signal includes the identity of the event and the corresponding actual parameters. Event signals are queued at the trigger in order of priority and responded to in turn.

Finally, it should be noted that a trigger can make both "synchronous" and "asynchronous" responses to events. The body of an **upon** statement (within the **do** ... **end** block) is executed synchronously with the event signal in the same sense that an **accept** statement is executed synchronously with an entry call. While the **upon** statement is executing the execution of the corresponding relation is suspended at the point at which the signal is generated (either acceptance or completion of the rendezvous for the relation entry). However, the trigger does not execute a full rendezvous with the relation, and no parameters or exceptions are returned from the trigger to the relation. Once the **upon** statement completes the synchronization with the relation is released and the trigger and relation proceed in parallel. A sequence of statements immediately following an **upon** statement thus executes asynchronously with the relation and can be used to provide an

```
with Source_to_Object, Word_Count;
with Code_Types; use Code_Types;


trigger Maintain_Source_WC is
    t1: src_to_obj_tuple;
begin
    loop
        select
            upon Source_to_Object.insert
                (src: in source_code) completion
            do
                Word_Count.insert(src);
            end upon;
        or
            upon Source_to_Object.update
                (t: in src_to_obj_tuple;
                update_src: in boolean => false;
                src: in source_code) completion
            do
                for t2 in Word_Count where
                    t2.text = t.src
                loop
                    Word_Count.delete (t2);
                    Word_Count.insert (src);
                end loop;
            end upon;
        or
            upon Source_to_Object.delete
                (t: in src_to_obj_tuple) completion
            do
                t1 := t;
            end upon;
            for t2 in Word_Count where
                t2.text = t.src
            loop
                Word_Count.delete (t2);
            end loop;
        end select;
    end loop;
end Maintain_Source_WC;
```

Figure 3: Trigger Maintain_Source_WC

```
with Source_to_Object, Word_Count;

predicate mandatory Source_Counted is
begin
    return
        every t1 in Source_to_Object satisfies
            some t2 in Word_Count satisfies
                t1.src = t2.text
            end some
        end every;
end Source_Counted;
```

Figure 4: Predicate Source_Counted

asynchronous response to relation operations. In `Maintain_Source_WC` the responses to `Source_to_Object` insert and update are made synchronously while the response to `Source_to_Object` delete is made asynchronously.

4.3 Predicates

Predicates are named boolean expressions over relations. Predicates, like functions, are program units. The expression language includes existentially and universally quantified forms and conditional expressions. An example predicate is shown in Figure 4.

The predicate states that for every tuple `t1` in relation `Source_to_Object` there is some tuple `t2` in relation `Word_Count` such that the `src` attribute of `t1` equals the `text` attribute of `t2`. In other words, there is a `Word_Count` tuple with the word-count results for every source-code object in `Source_to_Object`.

The **mandatory** keyword in the predicate declaration means that the predicate is required in every program that uses any of the relations to which the predicate refers. If the **mandatory** keyword is omitted, the predicate is only included in a program if explicitly imported using an Ada **with** clause.

As noted above, predicates are optionally enforcible. If a predicate is enforced, then no operation by a program on a relation designated by the predicate may leave the predicate violated. If the results of an

operation would violate an enforced predicate then they are rolled back and the exception `Constraint_Error` is raised. Each predicate has an associated boolean "attribute" (in the Ada sense) `enforced`. If this attribute is true, then predicate is enforced by default; otherwise, it is not. This attribute is assignable; thus the enforcement of a predicate can, in effect, be turned on and off. [1] (However, a mandatory predicate can also be declared **enforced** in which case it is always enforced by default and cannot be turned off.)

Overall, APPL/A provides an unusual degree of control over when and where predicates are enforced. Other aspects of predicates are described in [32].

4.4 Concurrency Control and Consistency Management

APPL/A includes five block-like statements that support concurrency control and consistency management with respect to relations. These include the **serial**, **atomic**, **suspend**, **enforce**, and **allow** statements. All excpet the **enforce** statement provide serializable access to a set of relations. The **suspend**, **enforce**, and **allow** statements affect the enforcement of predicates in various ways (either suspending or requiring the enforcement of designated predicates). The **atomic** and **suspend** statements may also entail rollback under some conditions (such as exception propagation or constraint violation). Each of the statements is discussed briefly below. An sketch of a program fragment showing the use of some of these statements is shown in Figure 5; this example is contrived, but it illustrates the syntax of the statements.

The **serial** statement provides simple *serializable* read or write access to designated relations. The **serial** statement has no special implications for predicate enforcement; predicates that are enforced in the immediately surrounding scope are enforced immediately within it. Neither does the **serial** statement offer rollback in the face of constraint violations or exceptions. Thus the **serial** statement is a unit of access to

[1] It is possible to make data inconsistent by turning on a predicate that was not previously enforced. However, consistency-management constructs such as the **suspend** statement (Section 4.4) can then be used to operate on the data even though they are inconsistent and to make them consistent if desired. In this way inconsistency is accommodated as a natural rather than exceptional condition.

```
serial read Word_Count;
begin
    - read Word_Count here
    ...
    serial write Source_to_Object;
    begin
        - operate on Source_to_Object here
        ...
        suspend Source_Counted;
        begin
            - add new source code to Source_to_Object,
            - temporarily violating Source_Counted;
            ...
            atomic write Word_Count;
            begin
                - update Word_Count atomically to include
                - new source code
                ...
            end atomic;
        end suspend;
    end serial;
end serial;
```

Figure 5: Sketch of Concurrency Control and Consistency Management Constructs

data but not a unit of completeness or consistency or work.

The **atomic** statement provides serializable and *recoverable* access to designated relations. Like the **serial** statement it carries no special significance for predicate enforcement, and in this respect it is not a unit of consistency. However, it imposes stronger requirements on completeness of work. If an **atomic** is terminated by the propagation of an exception (from any cause) then any operations performed on the designated relations are rolled back.

The **suspend** statement provides a unit of consistency with respect to predicate enforcement. It designates a list of predicates that are *not* to be enforced within its scope. This temporarily and locally overrides the default enforcement of those predicates, whatever it may be (but it does not affect the enforcement of those predicates in other processes or of other predicates in the same process). Serializable access is provided

to the relations named in the predicates. Operations that violate the suspended predicates are allowed to stand within the **suspend**. However, upon termination of the **suspend**, all predicates that are enforced in the surrounding scope must be satisfied; otherwise the **suspend** is rolled back. In contrast to the **atomic** statement, the **suspend** is not necessarily rolled back as a consequence of exception propagation; rollback occurs only if a predicate to be enforced is left unsatisfied. A **suspend** statement can be nested within an **atomic** statement to achieve the effects of a conventional transaction.

The **enforce** statement, unlike the **suspend** statement, *requires* rather than suspends the enforcement of designated predicates. Like the **suspend**, it locally and temporarily determines the actual enforcement of the designated predicates. There is no rollback associated with the **enforce** statement as a whole (although operations which violate enforced predicates are individually rolled back). Consequently, the **enforce** statement does not obtain serializable access to relations designated in the enforced predicates. (If serializable access is desired, the **enforce** statement can be nested within a **serial** statement.) By analogy with the **suspend** statement the **enforce** statement implies nothing about completeness of work, so there is similarly no special significance to the propagation of an exception.

The **suspend** statement enables transitions from one consistent state to another. The **enforce** statement enables transitions from an inconsistent to a consistent state. It seems inadvisable to offer a statement to move from a consistent to an inconsistent state. However, it can be useful to move from one inconsistent state to another, for example, to make a partial repair to a violated predicate. The **allow** statement offers a way to do this, provided that no new predicate violations are introduced. The **allow** statement names a set of predicates; if these are not satisfied upon entry to the **allow** then they need not be satisfied by operations in the body of the **allow** *or upon exit from it.* Within the **allow** operations that violate other enforced predicates are rolled back individually, but there is no rollback for the **allow** as a whole.

These statements allow processes to operate on data that are more or less constrained than desired. By providing serializable access they assure that interference by other processes can be precluded. By

enabling predicates to be locally enforced or suspended they allow a process to establish just the needed enforcement regime. By supporting rollback they enable erroneously incomplete or inconsistent work to be voided. These statements are more specialized in their effect than conventional database transactions, but in various combinations they can represent conventional transactions, nested transactions [18], and other capabilities for which there are not yet conventional names. This degree of flexibility in concurrency control and consistency management is unique to APPL/A.

5 Experience and Status

APPL/A is defined as an extension to Ada. The APPL/A definition [34] includes a formal syntax and English semantics with examples in a style similar to that of the Ada manual [1].

An earlier version of APPL/A [31] has been used to program several prototype process programs. One of these is REBUS, an executable system which supports the specification of software requirements in a functional hierarchy. REBUS maintains data about requirements in ten APPL/A relations; the relation specifications and bodies comprise about 2700 lines of source code, exclusive of runtime support systems and storage system interfaces. APPL/A was also used to extend REBUS to include features based on RSL/REVS [2,8] (using several more relations) and to construct a design support system based on the Rational Design Methodology of Parnas [21] and the IEEE design standard [7]. The experience gained with these process programs contributed greatly to our understanding of PPL requirements. This in turn motivated the revision of APPL/A to the definition described here. Another effort in the use of this earlier version of APPL/A is a process program to integrate software testing techniques [24]. This program is being rewritten using the version of APPL/A described here.

Work is presently underway on the design and implementation of an automatic translator for APPL/A. The translator is being built using existing Arcadia [36] Ada language technology. It will translate APPL/A programs into Ada. In addition, we are designing default implementations for APPL/A relations based

on some existing database systems such as Cactis [17] and Exodus [10]. (Previously, executable APPL/A programs have been generated by manual translation and implementation.)

Ongoing work is also aimed at developing prototype process programs in APPL/A to support the complete software life cycle. From these we hope to learn still more about PPL requirements and design.

6 Acknowledgements

This research was supported by the Defense Advanced Research Projects Agency, through DARPA Order #6100, Program Code 7E20, which was funded through grant #CCR-8705162 from the National Science Foundation. The authors wish to thank Deborah Baker, Roger King, Shehab Gamalel-din, Mark Maybee, Xiping Song, and Frank Belz for their advice. The comments of the members of the Arcadia consortium generally have also been important in clarifying the issues surrounding APPL/A.

References

[1] *Reference Manual for the Ada Programming Language.* United States Department of Defense, 1983. ANSI/MIL-STD-1815A-1983.

[2] Mack W. Alford. A requirements engineering methodology for real-time processing requirements. *IEEE Trans. on Software Engineering*, SE-3(1):60 – 69, January 1977.

[3] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA*, 1987.

[4] Malcolm P. Atkinson, Peter J. Bailey, K. J. Chisholm, W. P. Cockshott, and Ronald Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[5] Malcolm P. Atkinson and Peter O. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[6] Deborah A. Baker, David A. Fisher, and Jonathan C. Shultis. *Persistence and Type Integrity in a Software Development Environment.* Technical Report TR870702, Incremental Systems Corporation, 319 South Craig Street, Pittsburgh, PA 15213, July 1987.

[7] Jack H. Barnard, Robert F. Metz, and Arthur L. Price. A recommended practice for describing software designs: ieee standards project 1016. *IEEE Trans. on Software Engineering*, SE-12(2):258 – 263, February 1986.

[8] E. Bell, Thomas, David C. Bixler, and Margaret E. Dyer. An extendible approach to computer-aided software requirements engineering. *IEEE Trans. on Software Engineering*, SE-3(1):49 – 59, January 1977.

[9] Philip A. Bernstein. Database system support for software engineering – an extended abstract. In *Ninth International Conference on Software Engineering*, pages 166–178, ACM, 1987. To appear in ICSE9.

[10] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the exodus extensible dbms. In *Proc. of the International Workshop on Object Oriented Database Systems*, pages 52 – 65, 1986.

[11] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the exodus extensible database system. In *Proc. of the Twelfth International Conf. on Very Large Data Bases*, pages 91 – 100, 1986.

[12] Geoffrey M. Clemm. *The Odin System: an Object Manager for Extensible Software Environments.* Technical Report CU-CS-314-86, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado, 1986.

[13] Geoffrey M. Clemm and Leon J. Osterweil. *A Mechanism for Environment Integration.* Technical Report CU-CS-323-86, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, 1986.

[14] W. P. Cockshott, Malcolm P. Atkinson, K. J. Chisholm, Peter. J. Bailey, and Ronald Morrison. Persistent object management system. *Software – Practice and Experience*, 14:49–71, 1984.

[15] Don Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, March 1988.

[16] Dennis Heimbigner and Steven Krane. A graph transform model for configuration management environments. In *Proc. Third ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 216 – 225, November 1988. Special issue of SIGPLAN Notices, 24(2), February, 1989.

[17] Scott E. Hudson and Roger King. The cactis project: database support for software environments. *IEEE Trans. on Software Engineering*, 14(6):709–719, June 1988.

[18] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, Massachusetts Institute of Technology, May 1981.

[19] Jack A. Orenstein, Sunil K. Sarin, and Umeshwar Dayal. *Managing Persistent Objects in Ada: Final Technical Report.* Technical Report CCA-86-03, Computer Corporation of America, Cambridge, Massachusetts, May 1986.

[20] Leon J. Osterweil. Software processes are software too. In *Proc. Ninth International Conference on Software Engineering*, 1987.

[21] David L. Parnas and Paul C. Clements. A rational design process: how and why to fake it. *IEEE Trans. on Software Engineering*, SE-12(2):251 – 257, February 1986.

[22] Maria H. Penedo, Erhard Ploedereder, and Ian Thomas. Object management issues for software engineering environments – workshop report. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 226 – 234, ACM, 1988.

[23] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In *Proc. of the Fourteenth International Conf. on Very Large Data Bases*, pages 26 – 37, 1988.

[24] Debra Richardson, Stephanie Leif Aha, and Leon Osterweil. *Integrating Testing Techniques through Process Programming.* Technical Report 89-18, Department of Information and Computer Science, University of California, Irvine, Irvine, California, 92717, 1989.

[25] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proc. ACM SIGMOD Conf.*, pages 208–219, 1987.

[26] Lawrence A. Rowe and Michael R. Stonebraker. The POSTGRES data model. In *Proc. of the Thirteenth International Conf. on Very Large Data Bases*, pages 83 – 96, 1987.

[27] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *OOPSLA '86 Conf. Proc.*, pages 9–16, 1986. Available as ACM SIGPLAN Notices 21, 11, November 1986.

[28] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Reference Manual.* Technical Report DEC-TR-372, Digital Equipment Corporation, Hudson, Massachusetts, November 1985. version 1.1.

[29] John M. Smith, Steve Fox, and Terry Landers. *Reference Manual for ADAPLEX.* Technical Report CCA-83-08, Computer Corporation of America, May 1981.

[30] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proc. of the ACM SIGMOD International Conf. on the Management of Data*, pages 340 – 355, 1986.

[31] Stanley M. Sutton, Jr. *The APPL/A Programming Language – Background, Interim Definition, and Status.* Arcadia Document CU-88-11, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, October 1988.

[32] Stanley M. Sutton, Jr. Fcm: a flexible consistency model for software processes. October 1989. In preparation.

[33] Stanley M. Sutton, Jr. *Some Issues in the Design of 'Persistent Programming Languages'.* Arcadia Document CU-88-08, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, July 1988.

[34] Stanley M. Sutton, Jr. *Working Report on the Revised Definition for the APPL/A Programming Language.* Arcadia Document CU-89-05, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, June 1989.

[35] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. *Programmable Relations for Managing Change During Software Development.* Technical Report CU-CS-418-88, Univ. of Colorado, Dept. of Computer Science, Boulder, Colorado 80309, September 1988.

[36] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alexander Wolf, and Michal Young. Foundations for the arcadia environment architecture. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1 – 13, ACM, November 1988.